



Syddansk Universitet

A Core Model for Choreographic Programming

Cruz-Filipe, Luís; Montesi, Fabrizio

Published in:
Formal Aspects of Component Software

DOI:
[10.1007/978-3-319-57666-4_3](https://doi.org/10.1007/978-3-319-57666-4_3)

Publication date:
2017

Document version
Early version, also known as pre-print

Citation for pulished version (APA):
Cruz-Filipe, L., & Montesi, F. (2017). A Core Model for Choreographic Programming. In O. Kouchnarenko, & K. Ramtin (Eds.), Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers (pp. 17-35). Springer. Lecture Notes in Computer Science, Vol.. 10231, DOI: 10.1007/978-3-319-57666-4_3

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Choreographies, Computationally

Luís Cruz-Filipe

Department of Mathematics and Computer Science,
University of Southern Denmark
lcfilipe@gmail.com

Fabrizio Montesi

Department of Mathematics and Computer Science,
University of Southern Denmark
fmontesi@imada.sdu.dk

Abstract

We investigate the foundations of Choreographic Programming, a paradigm for writing concurrent programs that are deadlock free by construction, guided by the notion of computation. We start by introducing Minimal Choreographies (MC), a language that includes only the essential primitives of the paradigm. MC is minimal wrt Turing completeness: it implements all computable functions, and restricting its syntax breaks this property. Our methodology yields a natural notion of computation for choreographies, which can be used to generate concurrent implementations of independent computations automatically. Finally, we show that a Turing complete fragment of MC can be correctly projected to a process calculus (synthesis), which is thus both deadlock free and Turing complete.

1. Introduction

Choreographies are descriptions of concurrent systems that syntactically disallow writing mismatched I/O actions, adopting an “Alice and Bob” notation [29] for communications. An EndPoint Projection (EPP) can then be used to synthesise distributed implementations in process models, which are guaranteed to be deadlock-free by construction [9, 33]. We call this methodology *Choreographic Programming* [26]. Choreographies have been used in standards (e.g., BPMN [4] and WS-CDL [37]), language implementations [12, 19, 30, 35], type systems and logics [10, 11, 18]; they improve quality of software by making the desired communication behaviour explicit and easier to verify [2, 8, 9, 23].

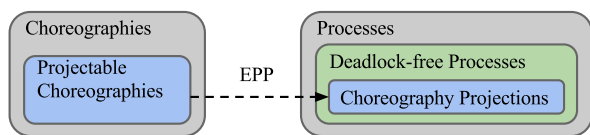


Figure 1. Choreographic Programming.

Typically, given a choreography language, only a subset of all choreographies can be projected correctly [2, 8, 23]. The EPPs of such projectable choreographies form a set of deadlock-free processes, which we call choreography projections. We depict this situation in Figure 1. As a consequence, a key aspect of any chore-

ography language model is understanding which concurrent programs are captured in the set of choreography projections, which are deadlock-free by construction. Following this direction, recent works proposed models for choreographic programming that include features of practical value in real-world scenarios, e.g., web services [8], multiparty sessions [9, 12], modularity [28], and run-time adaptation [31, 32].

As an emerging paradigm for concurrent programming, a precise characterisation of the computational power of choreographies is still missing: the expressivity of the aforementioned models is evaluated just by showing some examples. Despite the promise of the paradigm, we still know little of what can be computed with the code projected from choreographies, and what degree of parallelism such code exhibits. More generally, it is still unclear which features are essential in a choreography language to enable arbitrary computation, and none of the previously proposed models can be seen as canonical. How can one define a clear foundation for choreographic programming, such as λ -calculus is for functional programming? In this work, we present the first theoretical investigation of choreographies from a computational perspective, gaining insight on the set of choreography projections and the foundations of using choreographies as programs.

There are two major aspects that make our investigation of the computational power of implementation languages based on choreographies nontrivial. First, the choreography languages proposed so far are not minimal: they all come with differing domain-specific syntaxes (e.g., for channel mobility or runtime adaptation), and they include primitives for arbitrary local computation at each process that make these languages trivially Turing complete [8, 9, 28], because a single process can just run an arbitrary computation by itself. It is unclear how we can formalise a notion of computation based solely on the cornerstone feature of choreographies, i.e., communication structures arising from atomically-paired I/O actions. Second, EPP typically has many requirements, for example given in terms of complex type systems and syntactic conditions [8, 23]; therefore, it is difficult to understand precisely what processes can actually be correctly projected via EPP.

Our main contribution is a succinct characterisation of the computational power of choreography languages, which led us to the development of Minimal Choreographies, the first foundational model for understanding the paradigm of choreographic programming (writing concrete implementations using choreographies). We summarise our results below.

Minimal Choreographies. We introduce Minimal Choreographies (MC), a new choreography model designed to capture the essence of choreography languages (§ 2). MC features the key ingredients of choreography languages: processes (equipped with a local state) and communications between them. We restrict local computation at processes to three basic primitives on natural numbers (zero, successor, equality). MC is designed both to model the implementation of functions (Definition 1) and to capture concurrent behaviour

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CONF 'yy, Month d-d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-xxxx-xxxx-n/yy/mm...\$15.00.
<http://dx.doi.org/10.1145/nnnnnnnn.nnnnnnn>

$$\begin{aligned}
C &::= \eta; C \mid \text{if } p \stackrel{c}{=} q \text{ then } C_1 \text{ else } C_2 \mid \text{def } X = C_2 \text{ in } C_1 \mid X \mid \mathbf{0} \\
\eta &::= p.e \rightarrow q \mid p \rightarrow q[l] \quad e ::= \varepsilon \mid c \mid s \cdot c \quad l ::= L \mid R
\end{aligned}$$

Figure 2. Minimal Choreographies, Syntax.

(Definition 3), which will be the cornerstones of our construction. In § 3, we present a process model for modelling concurrent communications as in standard process calculi, the calculus of Stateful Processes (SP), and equip MC with a correct EPP procedure to SC (§ 3.3). Finally, we provide an amendment function (Definition 6) that, given any choreography, returns a projectable choreography (Lemma 2) performing the same data communications among processes. In other words, in MC the sets of choreographies and projectable choreographies are equivalent wrt the computations that they implement. This is the first time such a property is observed.

Functions as Choreographies. The notion of function implementation in MC keeps data separate from the processes that communicate and manipulate it. Following such notion, we show that all partial recursive functions, as defined by Kleene [21], can be implemented in (the projectable fragment of) MC (§ 4). MC is thus Turing complete.

Our proof is in line with other traditional proofs of computational completeness [14, 21, 36], where data and programs are distinct. This is different from the typical proofs of similar results for process calculi (e.g., π -calculus [34]) or the λ -calculus [1], which encode data as particular programs. The advantage in our setting is that our proof can be used in practice to construct choreographies that compute particular functions, as we exemplify in § 4.2. Our construction also parallelises sub-computations that are independent (Theorem 5), building upon the concurrent semantics of MC. By projecting our choreographies via EPP, we obtain corresponding function implementations in the process calculus SP (§ 4.3). As a corollary, we characterise a Turing complete and deadlock-free fragment of SP.

Implications of our results. In § 5, we show that MC is a core choreography language, in the sense that it is a minimal Turing complete language (§ 5.1) and that it can readily be embedded in previously presented models for choreographic programming (§ 5.2). The practical consequence of this, discussed in detail § 5.2, is the identification of MC as a simple common setting for the formal study of foundational questions in choreographies. This makes MC a good foundational model for choreographic programming, on the same level of, e.g., λ -calculus for functional programming and π -calculus for mobile processes.

2. Minimal Choreographies

In this section, we formally introduce Minimal Choreographies (MC). Programs in MC are choreographies that describe the behaviour of endpoint processes, which communicate by using explicit references to one another.

2.1 Syntax

The syntax of MC is reported in Figure 2, where C is a choreography. We make use of two (infinite) disjoint sets of names: processes (ranged over by p, q, \dots) and procedures (ranged over by X). The key idea in MC is that processes run in parallel to each other, and each process has a local memory cell that can be used to store a value – a string of the form $s \dots s \cdot \varepsilon$. Each process can access the value of its own memory cell using the placeholder c , but it cannot read the content of the memory cell of another process without performing a message exchange (no data sharing).

Term $\eta; C$ models an interaction among two processes, and reads “the system may execute the interaction η and proceed as C ”. An interaction η is either a value communication or a label selection. In a value communication $p.e \rightarrow q$, process p sends its local evaluation of expression e to process q , which stores the received value in its own memory cell. Expressions are restricted to be either the constant ε , the contents of the sender’s memory cell (written as c), or an application of the successor operator to c . In a selection $p \rightarrow q[l]$, process p communicates its selection of label l (which can be either L or R) to process q . In a conditional $\text{if } p \stackrel{c}{=} q \text{ then } C_1 \text{ else } C_2$, process q sends the content of its memory cell to process p , who checks if the received value is equal to that in its own cell; the choreography proceeds as C_1 , if that is the case, or as C_2 , otherwise. In all terms involving a sender and a receiver (value communications, selections, conditionals), their names must be different (no self-communications). The terms for the definition and the invocation of a recursive procedure X are standard. The term $\mathbf{0}$, also called *exit point*, is the terminated choreography.

2.2 Semantics

The semantics for MC uses reductions of the form $C, \sigma \rightarrow C', \sigma'$. The total function σ models the state of processes, mapping each process name to the value of its memory cell. We use v, w, \dots to range over values, defined as:

$$v, w, \dots ::= \varepsilon \mid s \cdot v$$

Values are isomorphic to natural numbers via $\lceil n \rceil = s^n \cdot \varepsilon$. The reduction relation \rightarrow is defined by the rules given in Figure 3. Rule $[C|Com]$ models a value communication $p.e \rightarrow q$. In the reductum, the state of the receiver is updated to the value v obtained by replacing the placeholder c in e with $\sigma(p)$ – the actual content of the sender’s cell. Rule $[C|Sel]$ is similar, but does not change σ . Rules $[C|Struct]$ and $[C|Ctx]$ are standard.

Rule $[C|Struct]$ uses *structural precongruence*, \preceq , which is the smallest precongruence satisfying the rules in Figure 4. We write $C \equiv C'$ for $C \preceq C'$ and $C' \preceq C$. In rule $[C|Unfold]$, we write $C_1[X]$ to indicate that the call term X occurs in C_1 , and replace it with the body of the recursive procedure on the right. In rules $[C|Eta-Eta]$, $[C|Eta-Cond]$ and $[C|Cond-Cond]$, we swap two terms describing actions performed by independent processes, modelling concurrent process execution; the auxiliary function $pn(C)$ returns the set of process names in C .

Remark 1 (Label Selection). *The reader unfamiliar with choreographies may wonder at this point about the role of label selection terms $p \rightarrow q[l]$ in MC, since their execution does not alter the state of any process. Selections are crucial in making choreographies projectable to distributed implementations in process models. We anticipate the intuition behind this point with a simple example. Consider the choreography:*

$$\text{if } p \stackrel{c}{=} q \text{ then } p.c \rightarrow r; \mathbf{0} \quad \text{else } r.c \rightarrow p; \mathbf{0}$$

Process p checks whether its value is the same as that of process q . If so, then process p communicates its value to process r ; otherwise, it is process r that communicates its value to p . Recall that we are interested in modelling processes that run independently and share no data. Here, the only process that knows which branch of the conditional should be executed is p , after comparing its own value with that received from q . However, process r also needs to know this information, since it must behave differently in the two branches. Intuitively, we have a problem because we are asking process r to act differently based on a decision made by another process, p , and there is no propagation of this decision from p to r (either directly or indirectly, through other processes). We can

$$\begin{array}{c}
\frac{v = e[\sigma(p)/c]}{p.e \rightarrow q; C, \sigma \rightarrow C, \sigma[q \mapsto v]} \text{ [C|Com]} \quad \frac{}{p \rightarrow q[l]; C, \sigma \rightarrow C, \sigma} \text{ [C|Sel]} \quad \frac{i = 1 \text{ if } \sigma(p) = \sigma(q), i = 2 \text{ o.w.}}{\text{if } p \stackrel{c}{=} q \text{ then } C_1 \text{ else } C_2, \sigma \rightarrow C_i, \sigma} \text{ [C|Cond]} \\
\\
\frac{C_1 \preceq C_2 \quad C_2, \sigma \rightarrow C'_2, \sigma' \quad C'_2 \preceq C'_1}{C_1, \sigma \rightarrow C'_1, \sigma'} \text{ [C|Struct]} \quad \frac{C_1, \sigma \rightarrow C'_1, \sigma'}{\text{def } X = C_2 \text{ in } C_1, \sigma \rightarrow \text{def } X = C_2 \text{ in } C'_1, \sigma'} \text{ [C|Ctx]}
\end{array}$$

Figure 3. Minimal Choreographies, Semantics.

$$\begin{array}{c}
\frac{\text{pn}(\eta) \cap \text{pn}(\eta') = \emptyset}{\eta; \eta' \equiv \eta'; \eta} \text{ [C|Eta-Eta]} \quad \frac{\{p, q\} \cap \text{pn}(\eta) = \emptyset}{\text{if } p \stackrel{c}{=} q \text{ then } (\eta; C_1) \text{ else } (\eta; C_2) \equiv \eta; \text{if } p \stackrel{c}{=} q \text{ then } C_1 \text{ else } C_2} \text{ [C|Eta-Cond]} \\
\\
\frac{}{\text{def } X = C_2 \text{ in } C_1[X] \preceq \text{def } X = C_2 \text{ in } C_1[C_2]} \text{ [C|Unfold]} \quad \frac{\text{pn}(C_i) \cap \text{pn}(\eta) = \emptyset}{\text{def } X = C_2 \text{ in } (\eta; C_1) \equiv \eta; \text{def } X = C_2 \text{ in } C_1} \text{ [C|Eta-Rec]} \\
\\
\frac{}{\text{def } X = C \text{ in } \mathbf{0} \preceq \mathbf{0}} \text{ [C|ProcEnd]} \quad \frac{\{p, q\} \cap \{r, s\} = \emptyset}{\begin{array}{c} \text{if } p \stackrel{c}{=} q \text{ then } (\text{if } r \stackrel{c}{=} s \text{ then } C_1 \text{ else } C_2) \text{ else } (\text{if } r \stackrel{c}{=} s \text{ then } C'_1 \text{ else } C'_2) \\ \equiv \\ \text{if } r \stackrel{c}{=} s \text{ then } (\text{if } p \stackrel{c}{=} q \text{ then } C_1 \text{ else } C'_1) \text{ else } (\text{if } p \stackrel{c}{=} q \text{ then } C_2 \text{ else } C'_2) \end{array}} \text{ [C|Cond-Cond]}
\end{array}$$

Figure 4. Minimal Choreographies, Structural precongurence \preceq .

easily fix the example by adding selections:

if $p \stackrel{c}{=} q$ then $p \rightarrow r[L]; p.c \rightarrow r; \mathbf{0}$ else $p \rightarrow r[R]; r.c \rightarrow p; \mathbf{0}$

Now, process p tells r about its choice via a label. Since the labels used in the two branches are different, r can infer the choice made by p from the label it receives.

The intuition about the role of label selections will be formalised in our definition of EndPoint Projection in 3.3. The first choreography we presented (without label selections) is not projectable, whereas the second one is.

MC enjoys the usual deadlock-freedom-by-design property of choreographies.

Theorem 1 (Deadlock-freedom by design). *If C is a choreography, then either:*

- $C \preceq \mathbf{0}$ (C has terminated);
- or, for all σ , $C, \sigma \rightarrow C', \sigma'$ for some C' and σ' (C can reduce).

Theorem 1 follows directly from the definition of our semantics. In § 3.3, we will use it to prove that the process implementations obtained by projecting choreographies is deadlock-free.

The semantics of MC also suggests a natural definition of computation.

Definition 1 (Function implementation in MC). *An actor choreography C implements a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ with input processes p_1, \dots, p_n and output process q if, for all $x_1, \dots, x_n \in \mathbb{N}$ and for every state σ s.t. $\sigma(p_i) = \ulcorner x_i \urcorner$:*

- if $f(\tilde{x})$ is defined, then $C, \sigma \rightarrow^* \mathbf{0}, \sigma'$ where $\sigma'(q) = \ulcorner f(\tilde{x}) \urcorner$;
- if $f(\tilde{x})$ is undefined, then $C, \sigma \not\rightarrow^* \mathbf{0}$.

Here, \rightarrow^* is the transitive closure of \rightarrow and $C, \sigma \not\rightarrow^* \mathbf{0}$ means that $C, \sigma \not\rightarrow^* \mathbf{0}, \sigma'$ for any σ' . By Theorem 1, in the second case C, σ must reduce infinitely (diverge).

Sequential composition and parallelism. Throughout this paper, we will be interested in processes with only one exit point (occurrence of $\mathbf{0}$). When a choreography C has a single exit point, we will write

$C \circledast C'$ for the choreography obtained by replacing the $\mathbf{0}$ subterm in C with C' . This does not add expressivity to MC, but it allows the usage of macros (as in the examples to come in § 2.3): $C \circledast C'$ behaves as a “sequential composition” of C and C' , as induction over C shows.

Lemma 1. *Let C be a choreography with a single exit point, C' be another choreography, and $\sigma, \sigma', \sigma''$ be states.*

1. If $C, \sigma \rightarrow^* \mathbf{0}, \sigma'$ and $C', \sigma' \rightarrow^* \mathbf{0}, \sigma''$, then $C \circledast C', \sigma \rightarrow^* \mathbf{0}, \sigma''$.
2. If $C, \sigma \not\rightarrow^* \mathbf{0}$, then $C \circledast C', \sigma \not\rightarrow^* \mathbf{0}$.
3. If $C, \sigma \rightarrow^* \mathbf{0}, \sigma'$ and $C', \sigma' \not\rightarrow^* \mathbf{0}$, then $C \circledast C', \sigma \not\rightarrow^* \mathbf{0}$.

The swap relation gives $C \circledast C'$ fully concurrent behaviour in some cases. Intuitively, C_1 and C_2 run in parallel in $C_1 \circledast C_2$ if their reduction paths to $\mathbf{0}$ can be interleaved in any possible way. Below, we write $C \xrightarrow{\tilde{\sigma}}^* \mathbf{0}$ for $C, \sigma_1 \rightarrow C_2, \sigma_2 \rightarrow \dots \rightarrow \mathbf{0}, \sigma_n$ where $\tilde{\sigma} = \sigma_1, \dots, \sigma_n$. We also write $\sigma(p)$ for the sequence $\sigma_1(p), \dots, \sigma_n(p)$.

Definition 2. *Let \tilde{p} and \tilde{q} be disjoint. Then, $\tilde{\sigma}$ is an interleaving of $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ wrt \tilde{p} and \tilde{q} if $\tilde{\sigma}$ can be partitioned into two subsequences $\tilde{\sigma}'_1$ and $\tilde{\sigma}'_2$ such that:*

- $\tilde{\sigma}'_1(p) = \tilde{\sigma}_1(p)$ for all $p \in \tilde{p}$, and $\tilde{\sigma}'_2(q) = \tilde{\sigma}_2(q)$ for all $q \in \tilde{q}$;
- $\tilde{\sigma}(r)$ is constant for all $r \notin \tilde{p} \cup \tilde{q}$.

Definition 3. *Let C_1 and C_2 be choreographies such that $\text{pn}(C_1) \cap \text{pn}(C_2) = \emptyset$ and C_1 has only one exit point. We say that C_1 and C_2 run in parallel in $C_1 \circledast C_2$ if: whenever $C_i \xrightarrow{\tilde{\sigma}_i}^* \mathbf{0}$, then $C_1 \circledast C_2 \xrightarrow{\tilde{\sigma}}^* \mathbf{0}$ for every interleaving $\tilde{\sigma}$ of $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ wrt $\text{pn}(C_1)$ and $\text{pn}(C_2)$.*

Theorem 2. *Let C_1 and C_2 be choreographies such that $\text{pn}(C_1) \cap \text{pn}(C_2) = \emptyset$ and C_1 has only one exit point. Then C_1 and C_2 run in parallel in $C_1 \circledast C_2$.*

This result is proved by induction over C_1 . Its converse is a consequence of (1) in Lemma 1.

2.3 Examples

We present some examples of MC choreographies. For presentational convenience, we define some macros (syntax shortcuts) using the notation $M(\text{params}) \triangleq C$, where M is the name of the macro, params its parameters, and C its body.

As a first example, we show how to increment the value stored by a process p , using an auxiliary process t that receives p 's value and replies with its successor.

$$\text{INC}(p, t) \triangleq p.c \rightarrow t; t.(s \cdot c) \rightarrow p; 0$$

Using INC , we build a choreography for addition. The macro $\text{ADD}(p, q, r, t_1, t_2)$ adds the values in p and q , storing the result in r , using auxiliary processes r, t_1 and t_2 . First, t_1 sets the value of r to zero, and then invokes the recursive procedure X . The idea in X is to increment the value of p as many times as the value in q (as in low-level abstract register machines). In the body of X , r checks whether its value is the same as q 's. If so, it informs the other processes that the recursion will terminate (selection of label L); otherwise, it asks the other processes to do another step (selection of label R). In each step, the values of p and r are incremented by using t_1 and t_2 as auxiliary processes. The compositional usage of INC is allowed, as it has exactly one exit point.

$$\begin{aligned} \text{ADD}(p, q, r, t_1, t_2) &\triangleq \\ \text{def } X = &\text{ if } r \stackrel{c}{=} q \text{ then } r \rightarrow p[L]; r \rightarrow q[L]; \\ &\quad r \rightarrow t_1[L]; r \rightarrow t_2[L]; 0 \\ &\text{ else } r \rightarrow p[R]; r \rightarrow q[R]; r \rightarrow t_1[R]; \\ &\quad r \rightarrow t_2[R]; \text{INC}(p, t_1) \circ \text{INC}(r, t_2) \circ X \\ &\text{ in } t_1.e \rightarrow r; X \end{aligned}$$

By Theorem 2, the calls to $\text{INC}(p, t_1)$ and $\text{INC}(r, t_2)$ can be executed in parallel. Indeed, using rule $[C|\text{Eta-Eta}]$ from Figure 4 repeatedly we obtain the following swapping:

$$\begin{aligned} &\underbrace{p.c \rightarrow t_1; t_1.(s \cdot c) \rightarrow p; r.c \rightarrow t_2; t_2.(s \cdot c) \rightarrow r; X}_{\text{expansion of } \text{INC}(p, t_1)} \quad \underbrace{}_{\text{expansion of } \text{INC}(r, t_2)} \\ &\preceq \underbrace{r.c \rightarrow t_2; t_2.(s \cdot c) \rightarrow r; p.c \rightarrow t_1; t_1.(s \cdot c) \rightarrow p; X}_{\text{expansion of } \text{INC}(r, t_2)} \quad \underbrace{}_{\text{expansion of } \text{INC}(p, t_1)} \end{aligned}$$

3. Stateful Processes (SP)

We now present the model of Stateful Processes (SP), the process calculus that we will use to generate process implementations from choreographies in MC. SP uses the same mechanism of direct process references for communications found in MC. Using this similarity, we give a natural definition of EndPoint Projection (EPP), which compiles choreographies in MC to terms in SP.

3.1 Syntax

The syntax of SP is reported below. Networks, ranged over by N, M , are either the inactive network 0 or parallel compositions of processes $p \triangleright_v B$, with p the process's name, v the value in its memory cell, and B its behaviour.

$$\begin{aligned} N, M &::= p \triangleright_v B \mid 0 \mid N \mid M \\ B &::= q!(e); B \mid p?; B \mid q \oplus l; B \mid p\&\{l_i : B_i\}_{i \in I} \mid 0 \\ &\quad \mid \text{if } c \stackrel{c}{=} q \text{ then } B_1 \text{ else } B_2 \mid \text{def } X = B_2 \text{ in } B_1 \mid X \end{aligned}$$

We comment on behaviours. Expressions and labels are as in MC. A send term $q!(e); B$ sends the evaluation of expression e to q , proceeding as B . Term $p?; B$, the dual receiving action, stores the value received from process p in the memory cell of the process executing the behaviour, proceeding as B . A selection term $q \oplus l; B$ sends label l to process q . These are received by the branching term

$p\&\{l_i : B_i\}_{i \in I}$, which can receive any of the labels l_i and proceed according to B_i . Branching terms offer either: a single branch with label L ; a single branch with label R ; or two branches with distinct labels L and R . In a conditional $\text{if } c \stackrel{c}{=} q \text{ then } B_1 \text{ else } B_2$, the process receives a value from another process q (synchronising with a send term) and compares it with its value to choose between the continuations B_1 and B_2 . The other terms are standard (definition of recursive procedures, procedure calls, and termination).

3.2 Semantics

The key difference between the semantics for MC and AP is that execution is now distributed, requiring synchronisation of processes executed in parallel.

Rule $[S|\text{Com}]$ follows the standard communication rule in process calculi. A process p executing a send action towards a process q can synchronise with a receive-from- p action at q ; in the reduct, q 's memory cell is updated with the value sent by p , obtained by replacing the placeholder c in e with the value from the memory cell at p . Rule $[S|\text{Sel}]$ is standard selection, as in session types [20], with the sender process selecting one of the branches offered by the receiver. In rule $[S|\text{Cond}]$, process p (executing the conditional) acts as a receiver for the value sent by the process whose value it wants to read (q). All other rules are standard; rule $[S|\text{Struct}]$ uses the structural precongruence \preceq , which is the smallest precongruence satisfying commutativity of \mid and the rules below.

$$\begin{aligned} \frac{}{N \mid p \triangleright_v 0 \preceq N} [S|\text{PZero}] \quad \frac{}{N \mid 0 \preceq N} [S|\text{NZero}] \\ \frac{}{\text{def } X = B \text{ in } 0 \preceq 0} [S|\text{ProcEnd}] \\ \frac{}{\text{def } X = B_2 \text{ in } B_1[X] \preceq \text{def } X = B_2 \text{ in } B_1[B_2]} [S|\text{Unfold}] \end{aligned}$$

As for MC, we can define function implementation in SP.

Definition 4 (Function implementation in SP). *An actor network N implements a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ with input processes p_1, \dots, p_n and output process q if $N \preceq (\prod_{i \in [1, n]} p_i \triangleright_{v_i} B_i) \mid q \triangleright_w B' \mid N'$ and, for all $x_1, \dots, x_n \in \mathbb{N}$:*

- if $f(\tilde{x})$ is defined, then $N(\tilde{x}) \rightarrow^* q \triangleright_{f(\tilde{x})} 0$;
- if $f(\tilde{x})$ is not defined, then $N(\tilde{x}) \not\rightarrow^* 0$.

where $N(\tilde{x})$ is a shorthand for $N[\widetilde{v_i} / v_i]$, i.e., the network obtained by replacing in N the values of the input processes with the arguments of the function.

3.3 Endpoint Projection (EPP)

We first discuss the rules for projecting the behaviour of a single process p , a partial function $\llbracket C \rrbracket_p$, given in Figure 6. All rules follow the intuition of projecting, for each choreography term, the local action performed by the process that we are projecting. For example, for a communication term $p.e \rightarrow q$, we project a send action for the sender process p , a receive action for the receiver process q , or just the continuation otherwise. The rule for selection is similar.

The rules for projecting recursive definitions and calls assume that procedure names have been annotated with the process names appearing inside the body of the procedure in order to avoid projecting unnecessary procedure code (see [8]).

The rule for projecting a conditional is more involved. In particular, we use the (partial) merging operator \sqcup from [8] to merge the behaviour of a process that does not know which branch has been chosen yet: $B \sqcup B'$ is isomorphic to B and B' up to branching, where the branches of B or B' with distinct labels are also included. As an example, consider the following choreography C :

$$C = \text{if } p \stackrel{c}{=} q \text{ then } p \rightarrow r[L]; C_1 \text{ else } p \rightarrow r[R]; C_2$$

$$\begin{array}{c}
\frac{u = e[v/c]}{p \triangleright_v q!\langle e \rangle; B_1 \mid q \triangleright_w p?; B_2 \rightarrow p \triangleright_v B_1 \mid q \triangleright_w B_2} \text{[S|Com]} \quad \frac{B_1 \rightarrow B'_1}{p \triangleright_v \text{def } X = B_2 \text{ in } B_1 \rightarrow p \triangleright_v \text{def } X = B_2 \text{ in } B'_1} \text{[S|Ctx]} \\
\\
\frac{j \in I}{p \triangleright_v q \oplus l_j; B \mid q \triangleright_w p\&\{l_i : B_i\}_{i \in I} \rightarrow p \triangleright_v B \mid q \triangleright_w B_j} \text{[S|Sel]} \quad \frac{N \rightarrow N'}{N \mid M \rightarrow N' \mid M} \text{[S|Par]} \\
\\
\frac{i = 1 \text{ if } v = e[w/c], \quad i = 2 \text{ otherwise}}{p \triangleright_v \text{if } c \stackrel{\leq}{=} q \text{ then } B_1 \text{ else } B_2 \mid q \triangleright_w p!\langle e \rangle; B' \rightarrow p \triangleright_v B_i \mid q \triangleright_w B'} \text{[S|Cond]} \quad \frac{N \preceq M \quad M \rightarrow M' \quad M' \preceq N'}{N \rightarrow N'} \text{[S|Struct]}
\end{array}$$

Figure 5. Stateful Processes, Semantics.

$$\begin{array}{c}
\llbracket \mathbf{0} \rrbracket_r = \mathbf{0} \quad \llbracket p.e \rightarrow q; C \rrbracket_r = \begin{cases} q!\langle e \rangle; \llbracket C \rrbracket_r & \text{if } r = p \\ p?; \llbracket C \rrbracket_r & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{o.w.} \end{cases} \quad \llbracket p \rightarrow q[l]; C \rrbracket_r = \begin{cases} q \oplus l; \llbracket C \rrbracket_r & \text{if } r = p \\ p\&\{l : \llbracket C \rrbracket_r\} & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{o.w.} \end{cases} \quad \llbracket X^{\tilde{p}} \rrbracket_r = \begin{cases} X & \text{if } r \in \tilde{p} \\ \mathbf{0} & \text{o.w.} \end{cases} \\
\\
\llbracket \text{if } p \stackrel{\leq}{=} q \text{ then } C_1 \text{ else } C_2 \rrbracket_r = \begin{cases} \text{if } c \stackrel{\leq}{=} q \text{ then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r & \text{if } r = p \\ p!\langle c \rangle; (\llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r) & \text{if } r = q \\ \llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r & \text{o.w.} \end{cases} \quad \llbracket \text{def } X^{\tilde{p}} = C_2 \text{ in } C_1 \rrbracket_r = \begin{cases} \text{def } X = \llbracket C_2 \rrbracket_r \text{ in } \llbracket C_1 \rrbracket_r & \text{if } r \in \tilde{p} \\ \llbracket C_1 \rrbracket_r & \text{o.w.} \end{cases}
\end{array}$$

Figure 6. Minimal Choreographies, Behaviour Projection.

The behaviour projection of C to process r is then $\llbracket C \rrbracket_r = p\&\{l : \llbracket C_1 \rrbracket_r, R : \llbracket C_2 \rrbracket_r\}$. If C did not include a selection from p to r , then r would not know which choice p had made in evaluating its condition. This aspect is found repeatedly in all choreography models [3, 8, 9, 18, 31, 33]. More specifically, while the originating choreography will execute correctly, its projection needs processes that behave differently in the branches of a conditional to be informed through a selection (either directly or indirectly, by receiving a selection from a previously notified process).

Rule [C|Sel] shows that a selection neither depends nor alters the execution state σ in any way, also in line with previous choreography calculi. Indeed, selection is not necessary for Turing completeness of MC (§ 4), but it is essential to characterise a Turing complete fragment of SP that can be projected from choreographies, since without it EPP cannot project interesting branching behaviour. This is because merging makes our projection partial; for example, the behaviour of process r cannot be projected in the following choreography because it does not know whether it should wait for a message from p or not.

$$C' = \text{if } p \stackrel{\leq}{=} q \text{ then } p.c \rightarrow r; \mathbf{0} \text{ else } \mathbf{0}$$

In this case, $\llbracket C' \rrbracket_r$ is undefined, as we cannot merge $p?; \mathbf{0}$ with $\mathbf{0}$ (the respective projections of the two branches for r).

Definition 5. Given C in MC and a state σ , the endpoint projection $\llbracket C, \sigma \rrbracket$ is the parallel composition of the processes in C : $\llbracket C, \sigma \rrbracket = \prod_{p \in \text{pn}(C)} p \triangleright_{\sigma(p)} \llbracket C \rrbracket_p$.

Since the σ s are total, if $\llbracket C, \sigma \rrbracket$ is defined for some σ , then $\llbracket C, \sigma' \rrbracket$ is defined for all other σ' . In this case, C is *projectable* and $\llbracket C, \sigma \rrbracket$ is the projection of C, σ .

Example 1. Given any σ , the EPP of $\text{INC}(p, t)$ defined above is:

$$\llbracket \text{INC}(p, t), \sigma \rrbracket = p \triangleright_{\sigma(p)} t!\langle c \rangle; t?; \mathbf{0} \mid t \triangleright_{\sigma(t)} p?; p!\langle s \cdot c \rangle; \mathbf{0}$$

Properties. EPP guarantees the following operational correspondence, the hallmark result of most formal choreography languages.

Theorem 3. Let C be a projectable choreography. Then, for all σ :

Completeness: If $C, \sigma \rightarrow C', \sigma'$, then $\llbracket C, \sigma \rrbracket \rightarrow^* \llbracket C', \sigma' \rrbracket$;

Soundness: If $\llbracket C, \sigma \rrbracket \rightarrow N$, then $C, \sigma \rightarrow C', \sigma'$ for some σ' , with $\llbracket C', \sigma' \rrbracket \prec N$.

Above, the *pruning relation* \prec [8, 9] eliminates branches introduced by the merging operator \sqcup when they are no longer needed to follow the originating choreography. We will abstract from \prec , since it does not alter the behaviour of a network: the eliminated branches are never selected, as shown in [8, 23, 31].

By Theorems 1 and 3, projections of MC terms never deadlock.

Corollary 1 (Deadlock-freedom by construction). Let $N = \llbracket C, \sigma \rrbracket$ for some C and σ . Then, either $N \preceq \mathbf{0}$ (N has terminated), or there exists N' such that $N \rightarrow N'$ (N can reduce).

Choreography Amendment. An interesting aspect of MC is that we can amend any unprojectable choreography to make it projectable, by adding some selections. Below, we assume that recursion variables are as for EPP and $\text{pn}(X^{\tilde{p}}) = \{\tilde{p}\}$.

Definition 6 (Amendment). Given C in MC, the transformation $\text{Amend}(C)$ repeatedly applies the following procedure until it is no longer possible, starting from the inner-most subterms in C . For each conditional subterm $\text{if } p \stackrel{\leq}{=} q \text{ then } C_1 \text{ else } C_2$ in C , let $\tilde{r} \subseteq (\text{pn}(C_1) \cup \text{pn}(C_2))$ be the largest set such that $\llbracket C_1 \rrbracket_{\tilde{r}} \sqcup \llbracket C_2 \rrbracket_{\tilde{r}}$ is undefined for all $r \in \tilde{r}$; then $\text{if } p \stackrel{\leq}{=} q \text{ then } C_1 \text{ else } C_2$ in C is replaced with:

$$\begin{array}{l}
\text{if } (p \stackrel{\leq}{=} q) \text{ then } (p \rightarrow r_1[L]; \dots; p \rightarrow r_n[L]; C_1) \\
\text{else } (p \rightarrow r_1[R]; \dots; p \rightarrow r_n[R]; C_2)
\end{array}$$

From the definitions of Amend, EPP and the semantics of MC, we get:

Lemma 2 (Amendment Lemma). Let C be a choreography. Then:

Completeness: $\text{Amend}(C)$ is defined;

Projectability: for all σ , $\llbracket \text{Amend}(C), \sigma \rrbracket$ is defined;

Correspondence: for all $\sigma, C, \sigma \rightarrow^* C', \sigma'$ iff $\text{Amend}(C), \sigma \rightarrow^* \text{Amend}(C'), \sigma'$.

Example 2. In the term if $(p \stackrel{c}{\approx} q)$ then $(p.c \rightarrow r; 0)$ else 0 , process r is not projectable. Its amendment is:

$$\text{if } (p \stackrel{c}{\approx} q) \text{ then } (p \rightarrow r[L]; p.c \rightarrow r; 0) \text{ else } (p \rightarrow r[R]; 0).$$

Thanks to merging, amendment also recognises some situations where selections are not needed. Consider the choreography $C = \text{if } p \stackrel{c}{\approx} q \text{ then } p.(s.c) \rightarrow r; 0 \text{ else } p.c \rightarrow r; 0$. Here, r does not need to know the choice made by p , since it always performs the same action (an input). Indeed, $\text{Amend}(C) = C$, and the projection for r is just $\llbracket \text{Amend}(C) \rrbracket_r = \llbracket C \rrbracket_r = p?; 0$.

4. Turing completeness of MC and SP

We now show that MC is Turing complete by using Kleene's well-known notion of *partial recursive function* [21]. We briefly review this formalism, and then show that every partial recursive function is implementable in MC. As a corollary, we characterise a deadlock-free fragment of SP that is also Turing complete.

4.1 Partial Recursive Functions

The class of partial recursive functions \mathcal{R} is inductively defined as follows.

Unary zero: $Z \in \mathcal{R}$, where $Z : \mathbb{N} \rightarrow \mathbb{N}$ is such that $Z(x) = 0$ for all $x \in \mathbb{N}$.

Unary successor: $S \in \mathcal{R}$, where $S : \mathbb{N} \rightarrow \mathbb{N}$ is such that $S(x) = x + 1$ for all $x \in \mathbb{N}$.

Projections: If $n \geq 1$ and $1 \leq m \leq n$, then $P_m^n \in \mathcal{R}$, where $P_m^n : \mathbb{N}^n \rightarrow \mathbb{N}$ satisfies $P_m^n(x_1, \dots, x_n) = x_m$ for all $x_1, \dots, x_n \in \mathbb{N}$.

Composition: if $f, g_i \in \mathcal{R}$ for $1 \leq i \leq k$, with each $g_i : \mathbb{N}^n \rightarrow \mathbb{N}$ and $f : \mathbb{N}^k \rightarrow \mathbb{N}$, then $h = C(f, \vec{g}) \in \mathcal{R}$, where $h : \mathbb{N}^n \rightarrow \mathbb{N}$ is defined by composition from f and g_1, \dots, g_k as: $h(\vec{x}) = f(g_1(\vec{x}), \dots, g_k(\vec{x}))$.

Primitive recursion: if $f, g \in \mathcal{R}$, with $f : \mathbb{N}^n \rightarrow \mathbb{N}$ and $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$, then $h = R(f, g) \in \mathcal{R}$, where $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ is defined by primitive recursion from f and g as: $h(0, \vec{x}) = f(\vec{x})$ and $h(x_0 + 1, \vec{x}) = g(x_0, h(x_0, \vec{x}), \vec{x})$.

Minimization: If $f \in \mathcal{R}$, with $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, then $h = M(f) \in \mathcal{R}$, where $h : \mathbb{N}^n \rightarrow \mathbb{N}$ is defined by minimization from f as: $h(\vec{x}) = y$ iff (1) $f(\vec{x}, y) = 0$ and (2) $f(\vec{x}, y)$ is defined and different from 0 for all $z < y$.

Our definition is slightly simplified, but equivalent to, that in [21], where it is also shown that \mathcal{R} is the class of functions computable by a Turing machine. In the remainder, we view the elements of \mathcal{R} intensionally, e.g. the functions Z and $C(Z, S)$ are distinct, although they always return the same value (zero).

Example 3 (Addition and Subtraction). We show that $\text{add} \in \mathcal{R}$, where $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$ adds its two arguments. As $\text{add}(0, y) = y$ and $\text{add}(x+1, y) = \text{add}(x, y) + 1$, we can define add by recursion as $\text{add} = R(P_1^1, C(S, P_2^2))$. Indeed, the function $y \mapsto \text{add}(0, y)$ is simply P_1^1 , whereas $1 + \text{add}(x, y)$ is $h(x, \text{add}(x, y), y)$ where $h(x, y, z) = y + 1$, which is the composition of the successor function with P_2^3 .

From addition, we can define subtraction by minimization, since $\text{sub}(x, y) = x - y$ is the smallest z such that $y + z = x$ (subtraction is not defined if $y > x$). We use an auxiliary function $\text{eq}(x, y)$ that returns 0 if $x = y$ and a non-zero value otherwise, which is known to be partial recursive. Then we can define subtraction as $\text{sub} = M(C(\text{eq}, C(\text{add}, P_2^3, P_3^3), P_1^3))$. Indeed, composing add with P_2^3 and P_3^3 produces $(x, y, z) \mapsto y + z$, and the outer composition yields $(x, y, z) \mapsto \text{eq}(y + z, x)$. This function evaluates to

0 precisely when $z = y - x$, and applying minimization computes this value from x and y .

4.2 Encoding Partial Recursive Functions in MC

All functions in \mathcal{R} can be implemented by a minimal choreography, in the sense of Definition 1. Given $f : \mathbb{N}^n \rightarrow \mathbb{N}$, its implementation is denoted $\llbracket f \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}}$; processes \vec{p} and \vec{q} are parameters. All choreographies we build have a single exit point, and we combine them using the sequential composition operator \S (§ 2.3).

Auxiliary processes are used for intermediate computation. Since MC does not provide primitives for generating fresh process names, we use r_0, r_1, \dots for these auxiliary processes, and annotate the encoding with the index ℓ of the first free auxiliary process name ($\llbracket f \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}}$). To alleviate the notation, we assign mnemonic names to these processes in the encoding and formalise their correspondence to the actual process names in the text. For the latter, let $\pi(f)$ be the number of auxiliary processes needed for encoding $f : \mathbb{N}^n \rightarrow \mathbb{N}$, defined by

$$\begin{aligned} \pi(S) &= \pi(Z) = \pi(P_m^n) = 0 \\ \pi(C(f, g_1, \dots, g_k)) &= \pi(f) + \sum_{i=1}^k \pi(g_i) + k \\ \pi(R(f, g)) &= \pi(f) + \pi(g) + 3 \\ \pi(M(f)) &= \pi(f) + 3 \end{aligned}$$

To simplify the presentation, we write \vec{p} for p_1, \dots, p_n (when n is already known) and $\{A_i\}_{i=1}^n$ for $A_1 \S \dots \S A_n$. Also, we do not include the selections needed to make our choreographies projectable, since these can be automatically inferred via our Amend procedure (we will formally use this aspect in § 4.3).

The encoding of the base cases is straightforward.

$$\begin{aligned} \llbracket Z \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}} &= p.\varepsilon \rightarrow q & \llbracket S \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}} &= p.(s.c) \rightarrow q \\ \llbracket P_m^n \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}} &= p_m.c \rightarrow q \end{aligned}$$

We now move to the inductive constructions. Composition is also simple. Let $h = C(f, g_1, \dots, g_k) : \mathbb{N}^n \rightarrow \mathbb{N}$. Then:

$$\llbracket h \rrbracket_{\ell}^{\vec{p} \mapsto \vec{q}} = \left\{ \llbracket g_i \rrbracket_{\ell_i}^{\vec{p} \mapsto \vec{r}'_i} \right\}_{i=1}^k \S \llbracket f \rrbracket_{\ell_{k+1}}^{\vec{r}'_1, \dots, \vec{r}'_k \mapsto \vec{q}}$$

where $\vec{r}'_i = r_{\ell+i-1}$, $\ell_1 = \ell + k$ and $\ell_{i+1} = \ell_i + \pi(g_i)$. Each auxiliary process r'_i connects the output of g_i to the corresponding input of f . Choreographies obtained inductively use these process names as parameters; name clashes are prevented by increasing ℓ . By definition of \S $\llbracket g_{i+1} \rrbracket$ is substituted for the (unique) exit point of $\llbracket g_i \rrbracket$, and $\llbracket f \rrbracket$ is substituted for the exit point of $\llbracket g_k \rrbracket$. The resulting choreography also has only one exit point (that of $\llbracket f \rrbracket$). In § 4.3 we discuss how to modify this construction slightly so that the g_i s are computed in parallel.

For the recursion operator, we need to use recursive procedures. Let $h = R(f, g) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$. Then, using the macro INC from § 2.3 for brevity:

$$\begin{aligned} \llbracket h \rrbracket_{\ell}^{\vec{p}_0, \dots, \vec{p}_n \mapsto \vec{q}} &= \text{def } T = \text{if } (r_c \stackrel{c}{\approx} p_0) \text{ then } (q'.c \rightarrow q; 0) \\ &\quad \text{else } \llbracket g \rrbracket_{\ell_g}^{r_c, q', p_1, \dots, p_n \mapsto r_t} \S \\ &\quad r_t.c \rightarrow q'; \text{INC}(r_c, r_t) \S T \\ &\text{in } \llbracket f \rrbracket_{\ell_f}^{p_1, \dots, p_n \mapsto q'} \S r_t.\varepsilon \rightarrow r_c; T \end{aligned}$$

where $q' = r_{\ell}$, $r_c = r_{\ell+1}$, $r_t = r_{\ell+2}$, $\ell_f = \ell + 3$ and $\ell_g = \ell_f + \pi(f)$. Process r_c is a counter, q' stores intermediate results, and r_t is a temporary storage cell; T checks the value of r_c and either outputs the result or recurs. The choreography has only one exit point (after the communication from r to q), since the exit points of $\llbracket f \rrbracket$ and $\llbracket g \rrbracket$ are replaced by code ending with calls to T .

The strategy for minimization is similar, but simpler. Let $h = M(f) : \mathbb{N}^n \rightarrow \mathbb{N}$. Again we use a counter r_c and compute

successive values of f , stored in q' , until a zero is found. This procedure may loop forever, either because $f(\tilde{x}, x_{n+1})$ is never 0 or because one of the evaluations itself never terminates.

$$\begin{aligned} \llbracket h \rrbracket_{\ell}^{p_1, \dots, p_{n+1} \mapsto q} = & \text{def } T = \llbracket f \rrbracket_{\ell_f}^{p_1, \dots, p_n, r_c \mapsto q'} \mathbin{\circ} r_c.\varepsilon \rightarrow r_z; \\ & \text{if } (r_z \stackrel{\leftarrow}{=} q') \text{ then } (r_c.c \rightarrow q; \mathbf{0}) \\ & \text{else } (\text{INC}(r_c, r_z) \mathbin{\circ} T) \\ & \text{in } r_z.\varepsilon \rightarrow r_c; T \end{aligned}$$

where $q' = r_\ell$, $r_c = r_{\ell+1}$, $r_z = r_{\ell+2}$, $\ell_f = \ell + 3$ and $\ell_g = \ell_f + \pi(f)$. In this case, the whole if-then-else is inserted at the exit point of $\llbracket f \rrbracket$; the only exit point of this choreography is again after communicating the result to q .

Definition 7. Let $f \in \mathcal{R}$. The encoding of f as a minimal choreography is $\llbracket f \rrbracket^{\tilde{p} \mapsto q} = \llbracket f \rrbracket_0^{\tilde{p} \mapsto q}$.

Example 4. We illustrate this construction by showing the encoding of the add and sub functions given in Example 3. Recall that $\text{add} = R(P_1^1, C(S, P_2^3))$. Expanding $\llbracket \text{add} \rrbracket^{p_x, p_y \mapsto q}$ we obtain:

$$\begin{aligned} \text{def } T = & \text{if } (r_1 \stackrel{\leftarrow}{=} p_x) \text{ then } (r_0.c \rightarrow q; \mathbf{0}) \\ & \text{else } \underbrace{r_0.c \rightarrow r_3}_{\llbracket P_2^3 \rrbracket_4^{r_1, r_0, p_y \mapsto r_3}} \mathbin{\circ} \underbrace{r_3.(s \cdot c) \rightarrow r_2}_{\llbracket S \rrbracket_4^{r_3 \mapsto r_2}} \\ & \quad r_2.c \rightarrow r_0; r_1.c \rightarrow r_2; r_2.(s \cdot c) \rightarrow r_1; T \\ & \quad \underbrace{\text{INC}(r_1, r_2)} \\ & \text{in } \underbrace{p_y.c \rightarrow r_0; r_2.\varepsilon \rightarrow r_1; T}_{\llbracket P_1^1 \rrbracket_3^{p_y \mapsto r_0}} \end{aligned}$$

The first two actions in the else branch are $\llbracket C(S, P_2^3) \rrbracket_3^{r_1, r_0, p_y \mapsto r_2}$.

For subtraction, we first show how to implement equality directly in MC, without resorting to its proof of membership in \mathcal{R} . This choreography is not the simplest possible because we want it to have only one exit point.

$$\begin{aligned} \text{EQ}(p_x, p_y, q, r) & \triangleq \text{def } T = (r.c \rightarrow q; \mathbf{0}) \text{ in} \\ & \text{if } (p_x \stackrel{\leftarrow}{=} p_y) \text{ then } (p_x.\varepsilon \rightarrow r; T) \text{ else } (p_x.(s \cdot c) \rightarrow r; T) \end{aligned}$$

Recall now that $\text{sub} = M(C(\text{eq}, C(\text{add}, P_2^3, P_3^3), P_1^3))$. Unfolding the encoding of minimization and composition, we obtain that $\llbracket \text{sub} \rrbracket_0^{p_x, p_y \mapsto q}$ is

$$\begin{aligned} \text{def } T = & \llbracket P_2^3 \rrbracket_7^{p_x, p_y, r_1 \mapsto r_5} \mathbin{\circ} \llbracket P_2^3 \rrbracket_7^{p_x, p_y, r_1 \mapsto r_6} \mathbin{\circ} \llbracket \text{add} \rrbracket_7^{r_5, r_6 \mapsto r_3} \mathbin{\circ} \\ & \llbracket P_1^3 \rrbracket_{11}^{p_x, p_y, r_1 \mapsto r_4} \mathbin{\circ} \text{EQ}(r_3, r_4, r_0, r_{11}) \mathbin{\circ} r_1.\varepsilon \rightarrow r_2; \\ & \text{if } r_2 \stackrel{\leftarrow}{=} r_0 \text{ then } (r_1.c \rightarrow q; \mathbf{0}) \text{ else } (\text{INC}(r_1, r_2) \mathbin{\circ} T) \\ & \text{in } r_2.\varepsilon \rightarrow r_1; T \end{aligned}$$

The first line in the definition of T is $\llbracket C(\text{add}, P_2^3, P_3^3) \rrbracket_5^{p_x, p_y, r_1 \mapsto r_3}$; the first five processes composed therewithin are

$$\llbracket C(\text{eq}, C(\text{add}, P_2^3, P_3^3), P_1^3) \rrbracket_3^{p_x, p_y, r_1 \mapsto r_0}.$$

Due to the way $\mathbin{\circ}$ works, this clear structure is lost when all definitions are unfolded.

4.3 Soundness

Structural induction on the construction of $\llbracket f \rrbracket^{\tilde{p} \mapsto q}$ shows that this construction is sound. The proof of this result is long, but not technically challenging (see Appendix).

Theorem 4 (Soundness). If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ and $f \in \mathcal{R}$, then, for every k , $\llbracket f \rrbracket_k^{\tilde{p} \mapsto q}$ implements f with input processes $\tilde{p} = p_1, \dots, p_n$ and output process q .

Let $\text{SP}^{\text{MC}} = \{\llbracket C, \sigma \rrbracket \mid \llbracket C, \sigma \rrbracket \text{ is defined}\}$ be the set of the projections of all projectable choreographies in MC. By Corollary 1 all terms in SP^{MC} are deadlock-free. By Theorems 3 and 4 and Lemma 2, SP^{MC} is also Turing complete.

Corollary 2. Every partial recursive function is implementable in SP^{MC} .

We finish this section with a comment on parallelism. If h is defined by composition from f and g_1, \dots, g_k , then in principle the computation of the g_i s could be completely parallelized. However, the encoding we gave does not fully achieve this, as their encodings share the processes containing the input.

Consider instead a modified variant $\{\!\{ \rrbracket \}$ of $\llbracket \rrbracket$ such that, for $h = C(f, g_1, \dots, g_k)$, $\{\!\{ h \rrbracket \}_{\ell}^{\tilde{p} \mapsto q}$ is

$$\left\{ p_j.c \rightarrow p_j \right\}_{1 \leq j \leq k, 1 \leq j \leq n} \mathbin{\circ} \left\{ \{\!\{ g_i \rrbracket \}_{\ell_i}^{\tilde{p}_i \mapsto r'_i} \right\}_{i=1}^k \mathbin{\circ} \{\!\{ f \rrbracket \}_{\ell_{k+1}}^{r'_1, \dots, r'_k \mapsto q}$$

with a suitably adapted label function ℓ . Now Theorem 2 applies, yielding:

Theorem 5. Let $h = C(f, g_1, \dots, g_k)$ and assume that $h(\tilde{x})$ is defined. For all processes \tilde{p} and q , if σ is a state such that $\sigma(p_i) = \ulcorner x_i \urcorner$, then the processes computing $g_i(\tilde{x})$ run in parallel in $\{\!\{ h \rrbracket \}_{\ell}^{\tilde{p} \mapsto q}$.

We could tweak our encoding, e.g., to obtain Turing completeness of MC using only a bounded number of processes, in the style of the two-counter machines in [25]. However, such constructions with bounded resources encode data using Gödel numbers, which is not in the spirit of our declarative notion of function implementation. They also restrict concurrency, breaking Theorem 5.

5. Minimality in Choreography Languages

We now briefly discuss our choice of primitives for Minimal Choreographies, showing that MC is indeed a minimal core language for choreographic programming. We first show that if we remove or simplify any of MC's primitives, we are no longer able to compute all partial recursive functions. Then, we discuss how MC can be embedded in fully fledged choreography languages presented in previous works.

5.1 Minimality of MC

We proceed by showing that removing or simplifying a primitive of MC yields a calculus with a decidable termination problem – and thus not Turing complete.

Basic primitives. The following constructs are trivially necessary.

- $\mathbf{0}$ (exit point): without this term, no choreography terminates.
- $p.e \rightarrow q$ (value communication): this is the only primitive that changes the content of a process. The syntax of expressions is heavily restricted, allowing only computation of the basic primitive recursive functions.
- $p \rightarrow q[l]$ (selection): this primitive adds no computational power to MC, but it is necessary for projectability. Without it, MC would still be Turing complete, but Corollary 2 would not hold. The set of labels is clearly minimal.
- $\text{def } X = C_2 \text{ in } C_1 \text{ and } X$ (recursion): without recursive definitions, each reduction decreases the number of operations in a choreography, so all choreographies terminate. Note that the recursion operator only allows tail recursion.

The only construct whose analysis is slightly more complex is the conditional.

Lemma 3. *Let C be a choreography with no conditionals. Then, termination of C is decidable and independent of the initial state.*

Proof. The second part is straightforward, since rule $[C|Cond]$ is the only rule whose conclusion depends on the state.

For the first part, we reduce termination to a decidable graph problem. Define $\mathcal{G}_C = \langle V, E \rangle$ to be the graph whose set of vertices V contains C and $\mathbf{0}$, and is closed under the following rules.

- if $\eta; C \in V$, then $C \in V$;
- if $\text{def } X = C_2 \text{ in } C_1 \in V$, then $C_1 \in V$;
- if $\text{def } X = C_2 \text{ in } \eta; C_1 \in V$, then $\text{def } X = C_2 \text{ in } C_1 \in V$;
- if $\text{def } X = C_2 \text{ in } \eta; X \in V$, then $\text{def } X = C_2 \text{ in } \eta; C_2 \in V$.

This set is finite: all rules add smaller choreographies to V , except the last one, which can only be applied once for each variable in C .

There is an edge between C_1 and C_2 iff $C_1, \sigma \rightarrow C_2, \sigma'$ for some σ, σ' without using rule $[C|Eta-Eta]$. This is decidable, as the possibility of a reduction does not depend on the state (as observed above). Also, if there is a reduction from C_1 , then there is always an edge from C_1 in the graph, as swapping communication actions cannot unblock execution.

Then C terminates iff there is a path from C to $\mathbf{0}$, which can be decided in finite time, as \mathcal{G}_C is finite. \square

Testing against fixed values also yields a decidable termination problem.

Lemma 4. *Let MC^v be the choreography calculus obtained from MC by replacing the conditional with if $p.c = v$ then C_1 else C_2 $i = 1$ if $\sigma(p) = v$, $i = 2$ otherwise and rule $[C|Cond]$ with $\frac{\text{if } p.c = v \text{ then } C_1 \text{ else } C_2, \sigma \rightarrow C_i, \sigma}{\text{Termination in } MC^v \text{ is decidable.}}$*

Proof. We first show that termination is decidable for processes of the form $\text{def } X = C_2 \text{ in } X$ and comparison with $\mathbf{0}$. The proof is by induction on the number of recursive definitions in C_2 .

Consider first the case where C_2 has no recursive definitions, and let P be the set of all process names occurring in C_2 . We define an equivalence relation on states by

$$\sigma \equiv_P \sigma' \text{ iff } (\forall p \in P, \sigma(p) = \varepsilon \text{ iff } \sigma'(p) = \varepsilon).$$

The vertices of the graph are the $2^{|P|}$ equivalence classes of states wrt \equiv_P , plus \top . Note that \equiv_P is compatible with the transition relation excluding rule $[C|Eta-Eta]$: for any choreography C using only process names in P , $\sigma_1 \equiv \sigma_2$ and $C, \sigma_i \rightarrow \sigma'_i$, then $\sigma'_1 \equiv \sigma'_2$.

The edges in the graph are defined as follows. There is an edge from $[\sigma]$ to $[\sigma']$ if $C_2, \sigma \rightarrow X, \sigma'$, and there is an edge from $[\sigma]$ to \top if $C_2, \sigma \rightarrow \mathbf{0}, \sigma'$ or $C_2, \sigma \rightarrow Y, \sigma'$ for some $Y \neq X$. This is constructible, as reductions in C_2 are always finite, and well-defined, as alternative reduction paths always end in the same state.

Since reductions are deterministic and \equiv_P is compatible with reduction, every node has exactly one edge leaving from it, except for \top . Therefore, we can decide if $\text{def } X = C_2 \text{ in } X$ terminates from an initial state σ by simply following the path starting at σ and returning Yes if we reach \top and No if we pass some node twice. This procedure terminates, as the graph is finite.

For the inductive step, proceed as above but add an extra node to the graph, labeled \perp . When constructing the edges in the graph, if C_2 reduces to a variable Y different than X , we split into two cases. If Y is not bound in C_2 , we proceed as in the previous case. If Y is bound, then we apply the induction hypothesis to the choreography $\text{def } Y = C_Y \text{ in } Y$ (where $Y = C_Y$ is the same as in C_2) to decide whether the reduction from Y will terminate; if this is not the case, we add an edge to \perp , otherwise we proceed with the simulation. At the end, we return No in the case that the path followed leads to \perp .

The general case follows, as C has the same behaviour as $\text{def } X = C \text{ in } X$ for some X not occurring in C .

If we allow comparisons with other values, the strategy is the same, but the relation \equiv_P has to be made finer. The key observation is that only a finite number of values can be used in comparisons, so we can identify states if they only differ on processes whose contents are larger than all values used in conditionals. \square

5.2 MC and other languages

We designed MC to be representative of the body of previous work on choreographic programming, where choreographies are used for implementations. Therefore, all the primitives of MC are either present or easily encodable in such languages, for example [8, 9, 12, 28, 31, 32, 37]. As a result, we obtain a notion of function implementation for these languages, induced by that for MC, for which they are Turing complete. For the calculus from [9], we report a formal translation from MC in Appendix B. In the following we give a brief overview of the significance of our results for the cited languages.

Differently from MC, other choreography languages typically use channel-based communications (as in the π -calculus [34]). Communications via process references as in MC can be easily encoded by assigning a dedicated channel to each pair of processes. For example, the calculus in [9], which we refer to as Channel Choreographies (CC), features an EPP targeting the session-based π -calculus [3]; CC is a fully-fledged calculus aimed at real-world application, and it has been implemented as a choreographic programming framework (the Chor language [12]). Our formal translation from MC to CC (given in Appendix B) shows that some primitives of CC are not needed to achieve Turing completeness, including: asynchronous communications, creation of sessions and processes, channel mobility, parameterised recursive definitions, arbitrary local computation, unbounded memory cells at processes, multiparty sessions. While these primitives are useful in practice, they come at the cost of making the formal treatment of CC very technically involved. In particular, CC (and its implementation Chor) requires a sophisticated type system, linearity analysis, and definition of EPP to ensure the correctness of projected processes. All these features are not needed in MC. Using our encoding from MC to CC, we can repeat the argument in § 4.3 to characterise a fragment of the session-based π -calculus from [3] that contains only deadlock-free terms and is Turing complete. CC has also been translated to the Jolie programming language [17, 27], whence our reasoning also applied to the latter and, in general, service-oriented languages based on message correlation.

The language WS-CDL from W3C [37] and the formal models inspired by it (e.g., [8]) are very similar to CC and a similar translation from MC could be formally developed, with similar implications as above. The same applies to the choreography languages developed in [31, 32], which add higher-order features to choreographies in terms of runtime adaptation. Finally, the language of compositional choreographies presented in [28] is an extension of CC and therefore our translation applies directly. This implies that adding modularity to choreographies does not add any computational power, as expected.

6. Related Work and Discussion

Unlimited Register Machines. The computational primitives in MC recall those of the Unlimited Register Machine (URM), a model of computation similar to computer hardware [14]. The URM stores natural numbers in memory cells, and has four basic operations on a cell's content: setting it to 0; increasing it; copying it to another cell; or comparing it to another cell's content and branching.

MC and URM differ in two main aspects. First, URM programs contain go-to statements (equivalent to general recursion), while MC supports only tail recursion. Second, the URM has centralised control: there is a single sequential program manipulating the cells. Instead, computation in MC is distributed among the various cells (the processes), which operate independently; thus, non-interfering interactions happen in parallel (see § 2.3 and Theorem 2).

Although the similarity of the URM and MC suggests that Theorem 4 could be proved by encoding the URM in MC, our proof using partial recursive functions is more direct. Thus, it gives us a (simple) algorithm to implement any function in MC, given its proof of membership in \mathcal{R} ; and it also yields the natural interpretation of parallelisation stated in Theorem 5.

Multiparty Sessions, Types, and Logics. Our primitives for communications in MC recall those used to describe protocols for multiparty sessions, e.g., in Multiparty Session Types (MPST) [3, 18] and conversation types [7]. These protocol descriptions are not meant for computation as our choreographies in MC; rather, they are types used to verify that sessions (e.g., π -calculus channels) are used accordingly to their respective protocol specifications. For such formalisms, we know of a strong characterisation result: a variant of MPST corresponds to communicating finite state machines [5] that respect the property of multiparty compatibility [16]. By contrast, for choreographies used as concrete implementations (our interest here), this question has barely been scratched before this work: session-typed choreographies with finite traces correspond to proofs in multiplicative-additive linear logic [10]. The language in [10] does not include any constructs for programming repetitive behaviour, e.g., recursion as in MC. To the best of our knowledge, MC is the first choreography language to be identified as minimally Turing complete.

Full β -reduction vs Swapping. The swapping of communications allowed by the prestructural congruence \preceq in MC makes the execution of choreographies nondeterministic. This recalls the notion of full β -reduction for the λ -calculus, which allows for sub-terms to be evaluated whenever possible. Despite this apparent similarity, the two mechanisms are different. Consider the choreography:

$$C \triangleq p.c \rightarrow q; q.\varepsilon \rightarrow r; 0$$

In the second communication, process q does not need to know which value it will receive from p in the first communication in order to proceed. Hence, we may imagine reducing C as follows for some σ , by allowing to consume the second communication before the first:

$$C, \sigma \rightarrow p.c \rightarrow q; 0, \sigma[r \mapsto \varepsilon]$$

This reduction follows the intuition of full β -reductions (reduce wherever possible), but it is actually disallowed by our semantics: rule $[C]_{\text{Eta-Eta}}$ cannot be applied because process q is present in both communications. The reason for this is that process identifiers play an important role in enforcing sequentiality: the choreography C clearly states that process q should *first* receive from p and then send to r . The behaviour projection for q elicits this:

$$[[C]]_q = p?; r!(\varepsilon); 0$$

Hence, the nondeterminism of choreographies can be controlled by the programmer using process identifiers, since the sequentiality constraints expressed for each process are respected by the semantics of MC and our EPP. This is a key practical feature of choreographies, as this kind of expressivity is important for the specification of interaction protocols and business processes among services [37]. For example, imagine that the choreography C models a payment transaction and that the message from q to r is a confirmation that p has sent its credit card information to q ; then, it is a natural requirement that the second communication happens only

after the first. Note that we would reach the same conclusions even if we adopted an asynchronous messaging semantics for SP, since the first action by q is a blocking input.

This kind of expressivity also makes choreographies challenging: combined with conditionals, it is what allows programmers to write unprojectable choreographies by using processes that do not know which branch to follow (Remark 1 in § 2). Fortunately, in MC all unprojectable choreographies can be fixed without changing the process identifiers in a choreography, but just by adding selections.

Nondeterminism. While the order in which communications are executed in MC can be nondeterministic due to swapping of communications, computation results are deterministic as in many other choreography languages [9, 10, 28]: if a choreography terminates, the result will always be the same regardless of how its execution is scheduled. This recalls the Church–Rosser Theorem for the λ -calculus [13].

We omitted nondeterministic computation because it is not necessary for our development. Nevertheless, due to its succinct definition and minimality, MC can be seen as promising stepping stone to explore primitives for expressing more kinds of concurrent behaviour. For example, it is easy to extend MC to support nondeterministic choices, borrowing from other developments for more abstract choreographies, such as in [6, 8, 23, 33]. Specifically, we could add the following primitive (we give syntax and semantics):

$$C ::= \dots | C_1 \oplus^p C_2 \quad \frac{i \in \{1, 2\}}{C_1 \oplus^p C_2 \rightarrow C_i} [C|\text{Choice}]$$

The choice $C_1 \oplus^p C_2$ reads as “process p (nondeterministically) chooses to proceed with C_1 or C_2 ”. Extending SP and our definition of EPP is straightforward, since a choice fundamentally behaves as an if-then-else conditional that nondeterministically chooses its continuation:

$$B ::= \dots | B_1 \oplus B_2 \quad \frac{i \in \{1, 2\}}{B_1 \oplus B_2 \rightarrow B_i} [P|\text{Choice}]$$

$$[[C_1 \oplus^p C_2]]_r = \begin{cases} [[C_1]]_r \oplus [[C_2]]_r & \text{if } r = p \\ [[C_1]]_r \sqcup [[C_2]]_r & \text{o.w.} \end{cases}$$

Amendment. In [24], the authors describe an amendment procedure for a choreography language that is not meant for implementations and is not Turing complete. Our amendment procedure is different since it is based on merging, which is not considered in [24].

We could define our amendment procedure in different ways, e.g., by propagating selections from a process to another as a chain, rather than from one process to all the others. This would not influence our results.

Extensions of MC. Our focus was to define a minimal version of MC ensuring Turing completeness. However, MC is an interesting basis for the development of a choreographic programming model based on direct process references, rather than channels. Many of the additional features given in the original presentation of CC could be ported to MC, e.g., the possibility to start new processes at runtime and asynchronous message queues [9]. We conjecture that all our results apply also to the asynchronous case, leaving these additions as future work.

Actors and Turing completeness. The fact that processes communicate by referring to the name of each other in our model recalls actor systems, where communications happen via process references, too. However, notable differences wrt how actor systems typically work are that communications in MC are synchronous and inputs specify the intended sender we wish to receive from. The first difference is just for the sake simplicity: it would be easy to introduce asynchrony in MC by using the technique exposed in [9]. The second difference arises because MC is a choreography calculus, and

communication primitives in choreographies typically express both sender and receiver.

Previous work investigated the expressive power of actor systems wrt computability [15]. The approach followed therein is based on the interplay between name restriction in process calculi and recursive procedures. Our development is still based on concurrent processes, but instead of name restrictions we use memory cells at processes, inspired by the URM, as mentioned above.

Acknowledgements

We thank Gianluigi Zavattaro for his useful comments. Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF-4005-00304 from the Danish Council for Independent Research.

References

- [1] Henk Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1985.
- [2] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Proc. of POPL*, pages 191–202. ACM, 2012.
- [3] Lorenzo Bettini, Mario Coppo, Loris D’Antoni, Marco De Luca, Mariangola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
- [4] Business Process Model and Notation. <http://www.omg.org/spec/BPMN/2.0/>.
- [5] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.
- [6] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition, SC 2007, Revised Selected Papers*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.
- [7] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.
- [8] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
- [9] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Proc. of POPL*, pages 263–274. ACM, 2013.
- [10] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In Paolo Baldan and Daniele Gorla, editors, *Proc. of CONCUR*, volume 8704 of *LNCS*, pages 47–62. Springer, 2014.
- [11] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In Luca Aceto and David de Frutos-Escrig, editors, *Proc. of CONCUR*, volume 42 of *LIPICs*, pages 412–426. Schloss Dagstuhl, 2015.
- [12] Chor. Programming Language. <http://www.chor-lang.org/>.
- [13] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.
- [14] Nigel J. Cutland. *Computability: an Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
- [15] Frank S. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. Decidability problems for actor systems. *Logical Methods in Computer Science*, 10(4), 2014.
- [16] Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Proc. of ICALP (II)*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
- [17] Maurizio Gabbriellini, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
- [18] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proc. of POPL*, pages 273–284. ACM, 2008.
- [19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Proc. of ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
- [20] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In Chris Hankin, editor, *Proc. of ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
- [21] Stephen C. Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Company, 1952.
- [22] Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. In *Proc. of CONCUR*, pages 395–409, 2013.
- [23] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In Antonio Cerone and Stefan Gruner, editors, *Proc. of SEFM*, pages 323–332. IEEE, 2008.
- [24] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In António Ravara and Josep Silva, editors, *Proc. of WWW 2013*, volume 123 of *EPTCS*, pages 34–48, 2013.
- [25] Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
- [26] Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf.
- [27] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014.
- [28] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In Pedro R. D’Argenio and Hernán C. Melgratti, editors, *Proc. of CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
- [29] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.
- [30] PI4SOA. <http://www.pi4soa.org>, 2008.
- [31] Mila Dalla Preda, Maurizio Gabbriellini, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In Tom Holvoet and Mirko Viroli, editors, *Proc. of COORDINATION*, volume 9037 of *LNCS*, pages 67–82. Springer, 2015.
- [32] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbriellini. AIOCI: A choreographic framework for safe adaptive distributed applications. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Proc. of SLE*, volume 8706 of *LNCS*, pages 161–170. Springer, 2014.
- [33] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proc. of WWW*, pages 973–982. ACM, 2007.
- [34] D. Sangiorgi and D. Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- [35] Savara. JBoss Community. <http://www.jboss.org/savara/>.
- [36] Alan M. Turing. Computability and λ -definability. *J. Symb. Log.*, 2(4):153–163, 1937.
- [37] W3C WS-CDL Working Group. Web services choreography description language version 1.0. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, 2004.

A. Proofs of Theorem 4 and Corollary 2

In the following, we use partial specifications of states. For example, $C, \{ p \mapsto v \} \rightarrow C', \{ q \mapsto w \}$ denotes that execution of C from any state where p contains value v will yield C' in some state where q contains value w .

Theorem 4. The proof is by induction on the definition of the set of partial recursive functions. We use a stronger induction hypothesis – namely, that if $\sigma(p_i) = \ulcorner x_i \urcorner$ and $f(\tilde{x})$ is defined, then $\llbracket f \rrbracket_k^{\tilde{p} \mapsto \tilde{q}}, \sigma \rightarrow^* \sigma'$ where $\sigma'(p_i) = \ulcorner x_i \urcorner$ and $\sigma'(q) = \ulcorner f(\tilde{x}) \urcorner$. The extra assumption that the input values are not changed during execution is essential for the inductive step.

1. For each base case, it is straightforward to compute the sequence of reductions from the rules and the definition of the corresponding actor choreography. We exemplify this with successor.

$$\llbracket S \rrbracket_{\ell}^{\tilde{p} \mapsto \tilde{q}} : p.(s \cdot c) \rightarrow q, \{ p \mapsto \ulcorner x \urcorner \} \rightarrow 0, \left\{ \begin{array}{l} p \mapsto \ulcorner x \urcorner \\ q \mapsto \ulcorner x + 1 \urcorner \end{array} \right\}$$

2. Let $h = C(f, g_1, \dots, g_k) : \mathbb{N}^n \rightarrow \mathbb{N}$. The result follows directly from the induction hypothesis and Lemma 1.
3. Let $h = R(f, g) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$. By induction hypothesis, choreographies $\llbracket f \rrbracket_{\ell_f}^{p_1, \dots, p_n \mapsto \tilde{q}}$ and $\llbracket g \rrbracket_{\ell_g}^{p_1, \dots, p_n+2 \mapsto \tilde{q}}$ implement f and g , respectively, for all \tilde{p}, q, ℓ_f and ℓ_g . Again, assume first that $h(x_0, \tilde{x})$ is defined. Then:

$$\begin{aligned} \llbracket h \rrbracket_{\ell}^{\tilde{p}_0, \tilde{p} \mapsto \tilde{q}} : \text{def } T &= (\dots) \text{ in } \llbracket f \rrbracket_{\ell_f}^{\tilde{p} \mapsto \tilde{q}'} ; \\ & r_t. \varepsilon \rightarrow r_c; T, \{ p_i \mapsto \ulcorner x_i \urcorner \} \\ \xrightarrow{IH}^* \text{def } T &= (\dots) \text{ in } r_t. \varepsilon \rightarrow r_c; T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner f(\tilde{x}) \urcorner \end{array} \right\} \\ & \rightarrow \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(0, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner 0 \urcorner \end{array} \right\} \end{aligned}$$

We now prove that

$$\begin{aligned} \text{def } T &= (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(k, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner k \urcorner \end{array} \right\} \\ & \rightarrow^* \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(k+1, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner k+1 \urcorner \end{array} \right\} \end{aligned}$$

for all $k < x_0$. We only need to unfold T once, so we omit the $\text{def } T = (\dots) \text{ in}$ wrapper in the next reduction sequence.

Since $k < x_0$, the definition of T reduces to the else branch:

$$\begin{aligned} T & \rightarrow^* \llbracket g \rrbracket_{\ell_g}^{r_c, q', \tilde{p} \mapsto r_t} ; r_t. c \rightarrow q'; \\ & r_c. c \rightarrow r_t; r_t.(s \cdot c) \rightarrow r_c; T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(k, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner k \urcorner \end{array} \right\} \end{aligned}$$

$$\begin{aligned} \xrightarrow{IH}^* & r_t. c \rightarrow q'; r_c. c \rightarrow r_t; r_t.(s \cdot c) \rightarrow r_c; \\ & T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(k, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner k \urcorner \\ r_t \mapsto \ulcorner g(k, h(k, \tilde{x}), \tilde{x}) \urcorner \end{array} \right\} \\ & \rightarrow^* T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(k+1, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner k+1 \urcorner \\ r_t \mapsto \ulcorner k \urcorner \end{array} \right\} \end{aligned}$$

which establishes the thesis, ignoring the value in r_t .

By induction on x_0 we obtain that

$$\begin{aligned} \llbracket h \rrbracket_{\ell}^{\tilde{p}_0, \tilde{p} \mapsto \tilde{q}}, \{ p_i \mapsto \ulcorner x_i \urcorner \} & \rightarrow^* \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(x_0, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner x_0 \urcorner \end{array} \right\} \\ \xrightarrow{(1)} \text{def } T &= (\dots) \text{ in } q'. c \rightarrow q; 0, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(x_0, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner x_0 \urcorner \end{array} \right\} \\ & \rightarrow^* \text{def } T = (\dots) \text{ in } 0, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ q' \mapsto \ulcorner h(x_0, \tilde{x}) \urcorner \\ r_c \mapsto \ulcorner x_0 \urcorner \\ q \mapsto \ulcorner h(x_0, \tilde{x}) \urcorner \end{array} \right\} \end{aligned}$$

and the last process is equivalent to 0 . In (1) we used the fact that the contents of r_c and p_0 are both equal to $\ulcorner x_0 \urcorner$.

If $h(x_0, \tilde{x})$ is not defined, there are two possible cases. If $f(\tilde{x})$ is not defined, then $\llbracket f \rrbracket_{\ell_f}^{\tilde{p} \mapsto \tilde{q}'}$ diverges from any state where each p_i contains $\ulcorner x_i \urcorner$, whence so does $\llbracket h \rrbracket_{\ell}^{\tilde{p}_0, \tilde{p} \mapsto \tilde{q}}$ by Lemma 1 and rule $[C|Ctx]$. If $g(k, h(k, \tilde{x}), \tilde{x})$ is undefined for some $k < x_0$, then divergence is likewise obtained from the fact that $\llbracket g \rrbracket_{\ell_g}^{r_c, q', \tilde{p}}$ diverges from any state where r_c contains $\ulcorner k \urcorner$, q' contains $\ulcorner h(k, \tilde{x}) \urcorner$, and p_i contains $\ulcorner x_i \urcorner$.

4. The case where $h = M(f) : \mathbb{N}^n \rightarrow \mathbb{N}$ is very similar, the auxiliary result now stating that

$$\begin{aligned} \text{def } T &= (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ r_c \mapsto \ulcorner k \urcorner \end{array} \right\} \\ & \rightarrow^* \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ r_c \mapsto \ulcorner k+1 \urcorner \end{array} \right\} \end{aligned}$$

as long as $f(\tilde{x}, k)$ is defined and different from 0.

The only new aspect is that non-termination may arise from the fact that $f(\tilde{x}, k)$ is defined and non-zero for every $k \in \mathbb{N}$, in which case we get an infinite reduction sequence

$$\begin{aligned} \llbracket h \rrbracket_{\ell}^{\tilde{p} \mapsto \tilde{q}}, \{ p_i \mapsto \ulcorner x_i \urcorner \} & \rightarrow^* \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ r_c \mapsto \ulcorner 0 \urcorner \end{array} \right\} \\ & \rightarrow^* \text{def } T = (\dots) \text{ in } T, \left\{ \begin{array}{l} p_i \mapsto \ulcorner x_i \urcorner \\ r_c \mapsto \ulcorner n \urcorner \end{array} \right\} \\ & \rightarrow^* \dots \quad \square \end{aligned}$$

Corollary 2. Let $f \in \mathcal{R}$. By Theorem 4, $C = \llbracket f \rrbracket^{\tilde{p} \mapsto \tilde{q}}$ for any suitable \tilde{p} and \tilde{q} implements f . By Lemma 2, $\text{Amend}(C)$ is projectable and operationally equivalent to C . Hence, by Theorem 3, $\llbracket \text{Amend}(C), \sigma \rrbracket$ is a term in SP that correctly implements f . \square

B. MC and Channel Choreographies

In this section, we present a formal translation from MC to the choreography calculus introduced in [9], which we here refer to as Channel Choreographies (CC). CC is designed to be projected to a variant of the session-typed π -calculus [3], which here we call Channel Processes (CP). Communications in CP are based on channels, instead of process names as in SP. This layer of indirection means that a process performing an I/O action does not know with which other process it is going to communicate, and that there can be race conditions on the usage of channels. CC comes with a typing discipline for checking that the usage of channels specified in a choreography will not cause errors in the process code generated by EPP.

B.1 Channel Choreographies (CC)

Syntax. We report the full syntax of CC, given in [9], but as we will see in § 5.2 some terms are unnecessary for our results;

$$\begin{aligned}
C &::= \eta; C \quad | \quad \text{if } p.(e = e') \text{ then } C_1 \text{ else } C_2 \quad | \quad \mathbf{0} \\
& \quad | \quad \text{def } X(\tilde{D}) = C_2 \text{ in } C_1 \quad | \quad X\langle \tilde{E} \rangle \quad | \quad (\nu r) C \\
\eta &::= \boxed{\widetilde{p[A]} \text{ start } \widetilde{q[B]} : a(k)} \quad | \quad p[A].e \rightarrow q[B].x : k \\
& \quad | \quad p[A] \rightarrow q[B] : k[l] \quad | \quad \boxed{p[A] \rightarrow q[B] : k\langle k'[C] \rangle} \\
D &::= p(\tilde{x}, \tilde{k}) \quad E ::= p(\tilde{e}, \tilde{k})
\end{aligned}$$

Figure 7. Channel Choreographies, Syntax.

we $\boxed{}$ such terms in our presentation of the syntax. In the original presentation of CC, expressions e may contain any basic values (integers, strings, etc.) or computable functions, making the language trivially Turing complete. Here, instead, we restrict them to the constant ε and the successor operator used on variables, i.e., $s \cdot x$. We also restrict labels l , originally picked from an infinite set, to be either L or R, as for MC. The major difference between MC and CC is the usage of *public channels* a and *session channels* k . Public channels are used to create new processes and channels at runtime, whereas session channels are used for point-to-point communications between processes. We will only need a single session channel in our development in § 5.2.

We comment on the syntax of CC, reported in Figure 7, where C is a channel choreography. An interaction η in CC can be either a start, a value communication, a selection, or a delegation. In a start term $\widetilde{p[A]} \text{ start } \widetilde{q[B]} : a(k)$, the processes \tilde{p} on the left synchronise at the public channel a in order to create a new private session k and spawn some new processes \tilde{q} (k and \tilde{q} are bound to the continuation). Each process is annotated with the role it plays in the created session. Roles are ranged over by A, B, C, \dots . They are used in the typing discipline of CC to check whether sessions are used according to protocol specifications, given as multiparty session types [18]. In a value communication $p[A].e \rightarrow q[B].x : k$, process p sends its evaluation of expression e over session k to process q , which stores the result in its local variable x ; the name x appearing under q is bound to the continuation. Differently from MC, where each process has only one memory cell accessed through the placeholder c , in CC each process has an unbounded number of cells (variables). Selections in CC, of the form $p[A] \rightarrow q[B] : k[l]$, are very similar to those in MC: the only difference is that we also have to write which role each process plays and the session used for communicating. In a delegation term $p[A] \rightarrow q[B] : k\langle k'[C] \rangle$, process p delegates its role C in session k' to process q ; delegation in CC is a typed form of channel mobility, inspired by the π -calculus.

In a conditional, process p chooses a continuation based on whether the expressions e and e' evaluate to the same value according to its own local state. The restriction term $(\nu r) C$ is standard and binds the scope of r (which can be either a process name p or a session channel name k) to C . Finally, in the definition of a recursive procedure, the parameters \tilde{D} indicate which processes are used in the body of the procedure and which variables and sessions are used by each process. In the invocation of a procedure $X\langle \tilde{E} \rangle$, each process can pass generic expressions as parameters to itself.

Semantics. The semantics of CC is given in terms of a reduction relation. We report the most interesting rules in Figure 8. Rule $[C|\text{Com}]$ is the key rule, where the value sent from a process p is received by a process q . Technically, this is modelled by replacing variable x with v in the continuation C , but only when it appears under the process name q (the *smart substitution* $C[v/x@q]$). Rule $[C|\text{Cond}]$ models an internal choice: p chooses a continuation

depending on whether the two values v and w are the same. The other rules are standard (recursion is treated similarly to MC).

The language CC offers the following deadlock-freedom-by-design property. Below, the structural precongruence \preceq for CC follows the same intuition as that for MC.

Theorem 6 (Deadlock-freedom-by-design in CC [9]). *Let C be a choreography with no free variable names. Then, either:*

- $C \preceq \mathbf{0}$ (C has terminated);
- or $C \rightarrow C'$ for some C' (C can reduce).

B.2 Channel Processes (CP)

We now present Channel Processes (CP), the target language that choreographies in CC can be projected to. We discuss only the terms used in our work (see [9] for a complete presentation).

Syntax. The (selected) syntax of processes (P, Q) is given below.

$$\begin{aligned}
P, Q &::= k[A]!B\langle e \rangle; P \quad | \quad k[B]?A(x); P \quad | \quad k[A]!B \oplus l; P \\
& \quad | \quad k[B]?A\&\{l_i : P_i\}_{i \in I} \quad | \quad \text{if } e = e' \text{ then } P \text{ else } Q \\
& \quad | \quad P \mid Q \quad | \quad \text{def } X(\tilde{x}, \tilde{k}) = Q \text{ in } P \quad | \quad X\langle \tilde{e}, \tilde{k} \rangle \quad | \quad \mathbf{0}
\end{aligned}$$

Binding occurrences are denoted by the usage of round parentheses. In term $k[A]!B\langle e \rangle; P$, as role A on session k , we send the value of expression e to B on the same session; then, we proceed as P . Dually, term $k[B]?A(x); P$ receives a message for role B from role A on session k and stores it in variable x . Terms $k[A]!B \oplus l$ and $k[B]?A\&\{l_i : P_i\}_{i \in I}$ model, respectively, branch selection and offering. The other terms are the standard parallel composition, procedure definition, procedure call, conditional (restricted to checking for equality), and terminated process.

Semantics. We discuss only the basic synchronous semantics of communications. All the other rules are standard (see [22]). As in typical calculi for multiparty sessions equipped with roles, each role in a session is a distinct communication endpoint. Therefore, a send action on a session k from a role A towards a role B synchronises with a receive action on the same session k by the target role B wishing to receive from the sender role A . For value and label communications, this intuition is formalised by the following rules:

$$k[A]!B\langle v \rangle; P \quad | \quad k[B]?A(x); Q \quad \rightarrow \quad P \quad | \quad Q[v/x]$$

$$k[A]!B \oplus l_j; P \quad | \quad k[B]?A\&\{l_i : Q_i\}_{i \in I} \quad \rightarrow \quad P \quad | \quad Q_j \quad (j \in I)$$

B.3 Endpoint Projection and Typing

As for MC, we present the Endpoint Projection (EPP) procedure for CC by defining first how to project the behaviour of a single process. The projection of a process p from a choreography C , written $\llbracket C \rrbracket_p$, is inductively defined on the structure of C in a similar way as the behaviour projection given in § 3.3 (see [9] for the full definition). We report the rules for projecting value communications and conditionals:

$$\llbracket p[A].e \rightarrow q[B].x : k; C \rrbracket_r = \begin{cases} k[A]!B\langle e \rangle; \llbracket C \rrbracket_r & \text{if } r = p \\ k[B]?A(x); \llbracket C \rrbracket_r & \text{if } r = q \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases}$$

$$\left[\begin{array}{l} \text{if } p.(e = e') \\ \text{then } C_1 \text{ else } C_2 \end{array} \right]_r = \begin{cases} \text{if } e = e' \\ \text{then } \llbracket C_1 \rrbracket_r \text{ else } \llbracket C_2 \rrbracket_r & \text{if } r = p \\ \llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r & \text{otherwise} \end{cases}$$

Above, the merging operator $P \sqcup Q$ works as in MC: it is isomorphic to P and Q aside from input branches with distinct labels, which are instead included in a larger input branching.

The complete EPP procedure from CC to CP is technically involved, because the start term $\widetilde{p[A]} \text{ start } \widetilde{q[B]} : a(k)$ found in CC

$$\frac{}{p[A].v \rightarrow q[B].x : k; C \rightarrow C[v/x@q]} [C|Com] \quad \frac{}{p[A] \rightarrow q[B] : k[l]; C \rightarrow C} [C|Sel] \quad \frac{i = 1 \text{ if } v = w, i = 2 \text{ otherwise}}{\text{if } p.(v = w) \text{ then } C_1 \text{ else } C_2 \rightarrow C_i} [C|Cond]$$

Figure 8. Channel Choreographies, Semantics (selected rules).

enables the reuse of the same services exposed at a public channel a for spawning processes with potentially different behaviour. However, since start terms and restriction of names are unnecessary for our development, we can use a much simpler definition.

Definition 8 (EPP from CC to CP [9]). *Given a choreography C , its EPP $\llbracket C \rrbracket$ is defined as:*

$$\llbracket C \rrbracket = \prod_{p \in \text{fp}(C)} \llbracket C \rrbracket_p$$

where $\text{fp}(C)$ returns the set of free process names in C .

B.4 Typing CC

Differently from MC, the EPP of a choreography in CC does not always yield correct results. Consider the following choreography:

$$C = p[A] \rightarrow q[B] : k[L]; q[B].\varepsilon \rightarrow p[A].x : k; r[A] \rightarrow q[B] : k[L]$$

The choreography C above always terminates by reaching $\mathbf{0}$. However, its EPP (albeit defined) may get stuck:

$$\llbracket C \rrbracket = \underbrace{\llbracket C \rrbracket_p}_{k[A]!B \oplus L; k[A]?B(x)} \mid \underbrace{\llbracket C \rrbracket_r}_{k[A]!B \oplus L} \mid \underbrace{\llbracket C \rrbracket_q}_{k[B]?A \& \{L : k[B]!A(\varepsilon); k[B]?A \& \{L : \mathbf{0}\}\}}$$

Above, we have a race between the projections of process p and process r for the selection of label L offered by process q . This is because both p and r play the same role A in session k and therefore the receiver (the projection of process q) cannot distinguish them. In the case where the race is won by the projection of process r , not only do we obtain a reduction not defined by the originating choreography, but we even get into a deadlocked situation:

$$\llbracket C \rrbracket \rightarrow k[A]!B \oplus L; k[A]?B(x) \mid k[B]!A(\varepsilon); k[B]?A \& \{L : \mathbf{0}\}$$

To avoid such situations, CC comes with a typing discipline based on multiparty session types that guarantees the absence of races.

A typing judgement for CC has the form $\Gamma; \Theta \vdash C \triangleright \Delta$, where Δ types the usage of sessions, Θ the ownership of roles by processes, and Γ variables and public channels.

Formally, the typing environment Γ contains variable typings of the form $x@p : S$, typing variable x at p with data type S (which can only be **nat** in our case). An environment Θ contains ownership typings of the form $p : k[A]$, read “process p owns role A in k ” (when writing $\Theta, p : k[A]$, it is assumed that no other process owns the same role for the same session in Θ). The environment Δ contains session typings of the form $k : G$, where G is a global type (from multiparty session types [18]). The syntax of global types is:

$$G ::= A \rightarrow B : \langle \text{nat} \rangle; G \mid A \rightarrow B : \{l_i : G_i\}_{i \in I} \mid \mu t; G \mid t \mid \text{end}$$

A global type G abstracts a communication between two roles in a session. A value communication is abstracted by $A \rightarrow B : \langle \text{nat} \rangle$ (we restrict values to be natural numbers). A global type $A \rightarrow B : \{l_i : G_i\}_{i \in I}$ allows any selection from A to B of one of the labels l_i , provided that then the session proceeds as specified by the corresponding continuation G_i . The other terms are for recursion (μt and t) and termination (end).

We discuss the most relevant typing rules for CC, given below.

$$\frac{\Gamma \vdash e@p : S \quad \Theta \vdash p : k[A], q : k[B] \quad \Gamma, x@q : S; \Theta \vdash C \triangleright \Delta, k : G}{\Gamma; \Theta \vdash p[A].e \rightarrow q[B].x : k; C \triangleright \Delta, k : A \rightarrow B : \langle S \rangle; G} [T|Com]$$

$$\frac{\Theta \vdash p : k[A], q : k[B] \quad j \in I \quad \Gamma; \Theta \vdash C \triangleright \Delta, k : G_j}{\Gamma; \Theta \vdash p[A] \rightarrow q[B] : k[l_j]; C \triangleright \Delta, k : A \rightarrow B : \{l_i : G_i\}_{i \in I}} [T|Sel]$$

$$\frac{\Gamma; \Theta \vdash C_1 \triangleright \Delta \quad \Gamma; \Theta \vdash C_2 \triangleright \Delta}{\Gamma; \Theta \vdash \text{if } p.(e = e') \text{ then } C_1 \text{ else } C_2 \triangleright \Delta} [T|Cond]$$

Rule $[T|Com]$ checks that, in a value communication on session k , the sender and receiver processes own their respective roles in session k ($\Theta \vdash p : k[A], q : k[B]$), that the protocol for session k expects a communication for their respective roles ($k : A \rightarrow B : \langle S \rangle; G$), and that the expression sent by the sender has the expected type S . Rule $[T|Sel]$ checks that a selection uses one of the labels expected by the protocol for the session ($j \in I$). Rule $[T|Cond]$ is standard, requiring both branches to have the same typing; observe that different communication behaviour in the two branches may still occur, because of rule $[T|Sel]$.

Well-typedness is preserved by reductions.

Theorem 7 (Subject Reduction [9]). *Let $\Gamma; \Theta \vdash C \triangleright \Delta$. Then $C \rightarrow C'$ implies that $\Gamma'; \Theta' \vdash C' \triangleright \Delta'$ for some Γ', Θ' and Δ' .*

Thanks to the type system of CC, we get an operational correspondence result for EPP from CC to CP:

Theorem 8 (Operational Correspondence (CC \leftrightarrow CP) [9]). *Let C be a well-typed channel choreography without start subterms (terms of the form $p[A] \text{ start } q[B] : a(k)$) and such that its endpoint projection $\llbracket C \rrbracket$ is defined. Then,*

- (Completeness) $C \rightarrow C'$ implies $\llbracket C \rrbracket \rightarrow \succ \llbracket C' \rrbracket$;
- (Soundness) $\llbracket C \rrbracket \rightarrow P$ implies $C \rightarrow C'$ and $\llbracket C' \rrbracket \prec P$.

where \prec is the pruning relation defined in [9].

Remark 2. In [9], Theorem 8 is more general: it covers also choreographies that may contain start terms. However, that result requires an additional static analysis on the usage of public channels in choreographies (linearity). We do not present linearity here since we do not need start terms for our development.

As for MC, by combining Theorem 8 with Theorem 6 we get that the EPP of a well-typed channel choreography never deadlocks:

Corollary 3 (Deadlock-freedom by construction for CC). *Let C be well-typed and $\llbracket C \rrbracket = P$. Then, either:*

- $P \preceq \mathbf{0}$ (P has terminated); or,
- there exists Q such that $P \rightarrow Q$ (P can reduce).

B.5 Embedding MC into CC

We now show how choreographies in MC can be embedded into the full-fledged CC. Our embedding provides an operational correspondence, which we combine with the properties of EPP for MC and CC.

The communication primitives of MC and CC are different. In MC, messages are passed directly between processes: each process knows whom it is sending to or receiving from in each communication step; in CC, communication is between roles in a session channel. To translate actor choreographies into channel choreographies, we therefore assign to each process a role syntactically identical to its name, and perform all communication over a fixed channel k .

Conditional terms are also not directly translatable, as CC evaluates guards in a single process. For this reason, each translated process uses two variables: x , storing its internal value, and y , used exclusively for temporary storage of a value required for a test.

Recall (§ 3.3) that $\text{pn}(A)$ returns the set of process names in A .

Definition 9 (Embedding of MC in CC). *The embedding of an actor choreography A in CC is $\{A\}$, inductively defined as follows.*

$$\begin{aligned} \{p.e \rightarrow q; A\} &= p[p].e[c/x] \rightarrow q[q].x : k; \{A\} \\ \{p \rightarrow q[l]; A\} &= p[p] \rightarrow q[q] : k[l]; \{A\} \\ \left\{ \text{if } p \stackrel{c}{=} q \text{ then } A \text{ else } A' \right\} &= q[q].x \rightarrow p[p].y : k; \\ &\quad \text{if } p.(x = y) \text{ then } \{A\} \text{ else } \{A'\} \\ \{\text{def } X = A \text{ in } A'\} &= (\text{def } X(*) = \{A\})[* / |A|] \text{ in } \{A'\} \\ \{X\} &= X(*) \quad \{0\} = 0 \end{aligned}$$

where $|A| = \{p(\{x, y\}, k) \mid p \in \text{pn}(A)\}$.

Lemma 5. $A \preceq A'$ if and only if $\{A\} \preceq \{A'\}$.

In order to compare the semantics of actor and channel choreographies, we need to take the state into account. This is done by viewing each state as a substitution, replacing all free occurrences of x with the actual content of the process it belongs to.

Definition 10 (Substitution induced by state). *Let A be an actor choreography and σ be a state. The substitution σ_A is defined as $\sigma_A = [x/\sigma(p)@p \mid p \in \text{pn}(A)]$, and the embedding of A in CC via σ is the channel choreography $\{A\}_\sigma = \sigma_A(\{A\})$.*

Below, \rightarrow^+ denotes a chain of one or more applications of \rightarrow .

Theorem 9 (Operational Correspondence (MC \leftrightarrow CC)). *Let A be an actor choreography. Then, for all σ :*

- (Completeness) $A, \sigma \rightarrow A', \sigma'$ implies $\{A\}_\sigma \rightarrow^+ \{A'\}_{\sigma'}$;
- (Soundness) $\{A\}_\sigma \rightarrow^+ C$ implies $A, \sigma \rightarrow^+ A', \sigma'$ and $C \rightarrow^* \{A'\}_{\sigma'}$.

Theorem 9 establishes the formal correspondence between MC and CC. Furthermore, we can use it to show Turing completeness of CC. Since the semantics of CC does not have state, the definition of implementation of a function is slightly different than that for MC.

Definition 11 (Implementation in CC). *A channel choreography C implements a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ with input variables $p_1.z_1, \dots, p_n.z_n$ and output variable $q.z$ if, for all $x_1, \dots, x_n \in \mathbb{N}$:*

- if $f(\tilde{x})$ is defined, then $C[z_i/\widetilde{f(\tilde{x})}^\top @ p_i] \rightarrow^* 0$, and q receives exactly one message with $\widetilde{f(\tilde{x})}^\top$ as the value transmitted;
- if $f(\tilde{x})$ is not defined, then $C[z_i/\widetilde{f(\tilde{x})}^\top @ p_i] \not\rightarrow^* 0$, and q never receives any messages.

Theorem 10 (Soundness). *If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is a partial recursive function, then $\{\llbracket f \rrbracket^{\widetilde{p} \mapsto q}\}$ implements f with input variables $\widetilde{p.x}$ and output variable $q.x$.*

We end our development by combining our results to characterise a Turing-complete and deadlock-free fragment of CP.

Let CP^{CC} be the smallest fragment of CP containing the projections of all typable and projectable choreographies in CC, formally: $\text{CP}^{\text{CC}} = \{\llbracket C \rrbracket \mid \llbracket C \rrbracket \text{ is defined}\}$. From Corollary 3, all terms in CP^{CC} are deadlock-free.

We now show that CP^{CC} is also Turing powerful. The development is similar to that for SP^{MC} (§ 4.3), but we need two additional steps. First, the operational correspondence theorem for the EPP of CC (Theorem 8) needs the projected channel choreography to be well-typed. Fortunately, this is always the case for the channel choreographies obtained by embedding amended MC terms.

Lemma 6. *Let A be an actor choreography and σ a state. Then, $C = \{\text{Amend}(A)\}_\sigma$ implies $\Gamma; \Theta \vdash C \triangleright \Delta$ for some Γ, Θ and Δ .*

Proof. Choosing Θ is trivial, as each process has its own role. For Γ , we assign type **nat** to all variables. Finally, $\Delta = k : G$, where G is inferred by abstracting the communications in C . The inductive construction of the latter is always possible since we applied Amend, so we can type each conditional with either the same global type or a branching global type with two labels. \square

Second, we need to know that the embedding of a projectable actor choreography is also projectable in CC.

Lemma 7. *If A is projectable, then $\{A\}_\sigma$ is projectable for any σ .*

Using these results, the proof of Corollary 2 for SP^{MC} can be adapted to yield the following property.

Corollary 4 (Turing completeness of CP^{CC}). *Every partial recursive function is implementable in CP^{CC} .*

We recap our development. We have defined suitable transformations that characterise operationally-equivalent fragments of choreography and process calculi based on direct process references (MC, SP) or indirect channels (CC, CP). We then have shown that such fragments are Turing complete, through a development that depends strictly on the global nature of choreographic descriptions.